ASPfun: A Functional and Distributed Object Calculus Semantics, Type-system, and Formalization

Ludovic Henrio, Florian Kammüller, Henry Sudhof

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

ASP_{fun}: A Functional and Distributed Object Calculus Semantics, Type-system, and Formalization

Ludovic Henrio — Florian Kammüller — Henry Sudhof

N° 6353

November 2007

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6353--FR+ENG

$\mathbf{ASP_{fun}}$: A Functional and Distributed Object Calculus Semantics, Type-system, and Formalization

Ludovic Henrio* , Florian Kammüller† , Henry Sudhof†

Thème COM — Systèmes communicants
Projet Oasis

Rapport de recherche n° 6353 — November 2007 — 18 pages

Abstract: Several paradigms exist for distributed computing, this paper tries to provide a sound foundation for autonomous objects communicating in a very structured way. We define $\mathrm{ASP_{fun}}$, a calculus of functional objects, behaving autonomously, and communicating by a request-reply mechanism: requests are method calls handled asynchronously, futures represent awaited results for requests, and replies return the result of a request to an object that holds the corresponding future.

This report first presents the $\mathrm{ASP_{fun}}$ calculus and its semantics. Secondly we provide a type system for $\mathrm{ASP_{fun}}$, which ensure the "progress" property: *while there is a request that is not reduced to a value, the computation can continue.* $\mathrm{ASP_{fun}}$ and its properties have been formalized and proved using the Isabelle theorem prover.

Key-words: Object calculus, futures, active objects, typing, theorem proving

* CNRS – I3S – INRIA, Sophia-Antipolis
† Technische Universität Berlin

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 —Télécopie : +33 4 92 38 77 65

$\mathbf{ASP_{fun}}$: Un calcul d'objets fonctionnel et distribué Semantique, typage, et formalisation

Résumé : Ce rapport présente un calcul d'objet distribué et sans dead-locks. Ce calcul repose sur un modèle à objets actifs, utilise la notion de futurs, et d'appel de méthode asynchrone entre objets actifs. Ce dosument présente aussi un système de types pour ce calcul. Le calcul, son système de type et ses propriétés ont été formaisés et démontrés dans l'assistant de preuve Isabelle/HOL

Mots-clés : Calculs d'objets, distribution, futurs, objets actifs, typage, preuves formelles

1 Introduction

We present here a formalisation of a functional active object language, featuring first class futures: objects are distributed into several activities; communications toward activities are asynchronous (remote) method calls; and futures are identifiers for the result of such asynchronous invocations. We call those futures "first class" because they can be transmitted between activities as any object.

Several distributed frameworks rely on some form of distributed objects and actors. Indeed, from the original actor paradigm [3], several languages have been designed. Some languages directly feature distributed active objects, like the ProActive library. Also several massively used distributed paradigms like Workflows, Web-Services, ... rely on some functional distributed objects/actors. In this work we aim at a formalization helpful for the understanding of these frameworks, and of the benefits brought by futures to these distributed languages.

Typing is a deeply studied technique for guaranteeing properties of programs [17]. This paper proves a classical typing properties: progress, in an unusual setting: distributed active objects.

We also believe that the formalisation of such a calculus in a theorem prover will be helpful in the future design of distributed languages, and can provide a reliable basis for proofs using paradigms such as distributed objects, futures, remote method invocations, actors or active objects.

To our mind, the main contributions of this paper are the following:

- A functional active object calculus with futures,
- A proposal for a type-system for active object languages,
- An investigation on how to provide a dead-lock free calculus featuring mono-threaded active objects and futures,
- A formalisation of those features in a theorem prover, that we expect reusable for investigating about futures, typing, or active objects paradigms.

This document is organized as follows. First based on a precise review on existing distributed languages and calculi, and their formalisation, we explain more precisely the objectives of this paper. Then Section 3 presents ASP_{fun} and its operational semantics. Section 4 provides a type system for ASP_{fun}, featuring two classical properties: subject-reduction and progress, implying the absence of dead-locks in the calculus. These properties have been proved using the Isabelle theorem prover as explained in Section 5.

2 Background and Objectives

It is not the purpose of this article to make an extensive review of concurrent calculi and languages, we focus here on the ones that to our mind better illustrate the impact of ASP_{fun}.

Actors [3] is a relatively widely used paradigm for distributed autonomous entities, and their interactions by messages. They are rather functional entities but their behaviour can be changed dynamically, giving them a state.

Futures have been studied several times in the programming languages literature, originally appearing in Multilisp [11] and ABCL [19]. Futures have been formalised in several settings, generally functional-based [15, 8, 10]; those developments rely on explicit creation of futures (via a future, or a thread creation primitive) in a concurrent but not distributed setting. ASP [4, 5] on the contrary is distribution oriented by the fact that there is no shared memory, and futures are created *transparently* upon a remote method invocation; moreover, ASP is based on an imperative object calculus: the $\varsigma_{\mathbf{imp}}$-calculus [1]. Explicit creation of futures are also features by the Java language for example.

This paper presents a complementary calculus to the preceding ones: a *functional distributed* object calculus with *transparent* futures. These futures can be passed around between different locations in a much transparent way; thanks to its functional nature, this calculus is deadlock free because a future can be returned before the end of its computation. However, the semantics of the calculus had to be carefully written in order to avoid operations that have a side effect.

More recently, in [7, 9], the authors suggest a communication model, called *AmbientTalk*, based on an actor-like language but with several queues for both sending and receiving messages whenever possible, i.e. whenever connected. Their communication model is quite similar to the ASP calculus presented in [5], but with queues for message sending, handlers invoked asynchronously and automatic asynchronous calls on futures; the resulting programming model is slightly different from ASP because there is no blocking synchronization in AmbientTalk. This programming model seems particularly adapted to loosely coupled small devices communicating over an ad-hoc network.

One contribution of this work is the formalization of the entire language, its semantics and type system plus the proof of safety and progress in an interactive theorem prover. We believe that in the disicipline of language development the application of mechanical verification is particularly relevant even if it comes at the prize of intensive and partly cumbersome work. Related works from the viewpoint of mechanized verification of related languages is Ciaffaglione's, Liquori's, and Miculan's formalization of the imperative extension of the ς-calculus in the theorem prover Coq most prominently using a coinductive definition and higher order abstract syntax [6]. However, they do not consider concurrency or distribution. With respect to concurrency, the formalization of the π-calculus in Isabelle/HOL by Roeckl is related. There, again higher order abstract syntax is employed. However, no objects are introduced. Ridge works on a formalization of concurrent OCaml in Isabelle/HOL. However, he concentrates on concurrency using abstraction techniques to improve automization of concrete algorithm proofs, and has not formalized objects at all.

What is new here is distribution. Not even considering the unusually high quality achieved by full mechanization, the originality of our approach lies in its distribution primitives.

Earlier work on a calculus of distributed objects is Jeffrey's calculus [13]. Jeffrey bases his distributed object calculus on Gordon and Hankin's concurrent object calculus by adding explicit locations in his extension. His main objective is to avoid configurations where one object at one location is being accessed by another. He enforces these restrictions by a type system. Apparently, already preservation is very diffcult to achieve because migrating objects can carry remote calls. By introducing serializable objects and corresponding types Jeffrey

finally arrives at a calculus and type system with subject reduction. Interestingly serializable objects are non-imperative. Compared to our calculus the most decisive difference is that we do not consider locations instead activities. The concept of futures explicitly supports remote access. In ASP_{fun} activities are all non-imperative, thus directly implementing the idea of serialization.

3 ASP_{fun}: A Functional Active Object Calculus

3.1 Prerequisite: The ς-calculus

The Theory of Objects consists in various ς-calculi that only consider objects and their manipulation as primitive [2]. The kernel calculus on which ASP_{fun} relies includes *object definition*, *method invocation*, and *method override*. An object consists of a set of labelled methods. A method is a function with one formal parameter that represents *self*, i.e., the object in which the method is contained. The ς-calculus relies on the following syntax.

$$\begin{array}{rll} a, b ::= & [l_j = \varsigma(x_j)b_j]^{j\in 1..n} & \text{object definition} \\ & \mid a.l_i & (i \in 1..n) \text{ method call} \\ & \mid a.l_i := \varsigma(x)b & (i \in 1..n) \text{ update} \end{array}$$

Object fields are not defined as they are considered as degenerate methods not using the self parameter. Therefore selection of a field or invocation (call) of a method are identical. Sigma calculus terms are identified modulo *renaming of variables* (*α-conversion*). For a better integration with the distributed calculus, we choose a small-step semantics ($\rightarrow_\varsigma$) for the ς-calculus as follows.

$$\text{CALL}\quad \frac{l_i \in \{l_j\}^{j\in 1..n}}{E\left[[l_j = \varsigma(x_j)b_j]^{j\in 1..n}.l_i\right] \rightarrow_\varsigma E\left[b_i\{x_i \leftarrow o\}\right]}$$

$$\text{UPDATE}\quad \frac{l_i \in \{l_j\}^{j\in 1..n}}{E\left[[l_j = \varsigma(x_j)b_j]^{j\in 1..n}.l_i := \varsigma(x)b\right] \rightarrow_\varsigma E\left[\left[l_i = \varsigma(x)b, l_j = \varsigma(x_j)b_j^{j\in(1..n)-\{i\}}\right]\right]}$$

In this semantics $E[t]$ denotes any term that contains the expression t. The term E is a term containing a single hole ($\bullet$), and $E[t]$ is the term obtained by replacing the single hole by t.

$$E ::= \quad \bullet \mid [l_i = \varsigma(x)E, l_j = \varsigma(x_j)b_j^{j\in(1..n)-\{i\}}] \mid E.l_i \mid E.l_i := \varsigma(x)a \mid a.l_i := \varsigma(x)E$$

3.2 Syntax

Like in ASP, one of the basic principle of ASP_{fun} is to perform a minimal extension of the syntax of ς-calculus. ASP_{fun} programs only use one additional primitive, *Active*, for creating an active object. The syntax for ASP_{fun} is:

$$\begin{array}{rll} a, b ::= & [l_j = \varsigma(x_j)b_j]^{j\in 1..n} & \text{object definition} \\ & \mid a.l_i & (i \in 1..n) \text{ method call} \\ & \mid a.l_i := \varsigma(x)b & (i \in 1..n) \text{ update} \\ & \mid \mathit{Active}(a) & \text{Active object creation} \end{array}$$

3.3 Informal Semantics

Informally, an active object is an object (ς-calculus term) in a location. Activating an object, $Active(a)$, means creating a new location with the object to be activated, a becomes an *active object*. An *activity* consist in such a location and contains an active object, and a set of tasks to be performed, this set of tasks is called *request queue*. Upon creation of the activity, the request queue is empty.

Every message sent toward this location is a method call to the activated object, such a *remote method invocation* is asynchronous: the effect of this method call is both to create a new request in the request queue of the destination and to replace the original method invocation by a reference to the result of the created request. Such a promised reply is called a *future*. in ASP_{fun}, futures are objects that can be passed to other objects, other requests, and other activities; several activities may have a reference to the same future. Trying to access to the content of a future (e.g. invoking a method on it) is a blocking operation, but any request value (even partially evaluated) can be returned: the value of the request replaces a reference to the corresponding future. This operation is called a *reply*, and allowing replies with a partially evaluated term avoids having deadlocks on future access.

ASP_{fun} is a highly concurrent language where reductions can occur anywhere, in any request of any activity. The only restriction on the reduction is that it is impossible for an object to be sent to another activity (e.g. an activity creation) if this object has free variables. It is difficult to give a natural semantics to the update of an active object, however, the functional nature of the calculus (updating an object creates a copy) oriented us toward the following semantics: a method update on an active object creates a new activity with the method updated.

Proving the deterministic nature of the calculus is out of the scope of this paper. Informally, depending on the execution the set of created activities and the number of requests may vary, however the result of the computation is always the same.

For example, evaluating $Active([l = \varsigma(x).a]).l)$ will first create an activity with the object $[l = \varsigma(x).a]$, then perform a remote invocation on the method l of this activity (which creates a future), and finally reply by replacing the future by the result of the invocation: the term a (possibly partially evaluated).

Practically, implementing strictly the semantics presented here is not very reasonable, because of its high concurrency, and the inefficiency of some execution path. However, we consider this work as a reliable basis for further studies on stateless objects, and as an interesting proposal for a semantics for autonomous interdependent entities, which in case they are stateless can be implemented such that they are deadlock free.

3.4 Small-Step Operational Semantics

First of all, the semantics requires to define two new sets of identifiers (in addition to the labels of ς-calculus): the activities (α,β,...) and the futures (f_i). The semantics of ASP_{fun} also necessitates to define some structures that will be used for the dynamic reduction, first of all we define a configuration as a (unordered) set of activities: a configuration is a mapping from activity identifiers

to activities. each activity is composed of a request queue (mapping from future identifiers to terms), and an active object (term). As futures are referenced from anywhere, two requests must correspond to two different futures; here, unicity is ensured by indexing them over disjoint families.

$$C ::= \alpha_i[(f_j \mapsto s_j)^{j\in I_i}\ ,\ t_i]^{i\in 1..p} \text{ where } \{I_i\} \text{ are disjoint subsets of } \mathbb{N}$$

We extended also the syntax for terms as dynamically, they may contain references to activities and futures:

$$\begin{array}{rrr} s,t ::= & a & \text{Static term, as defined above} \\ & \alpha & \text{active object reference} \\ & f_i & \text{future reference} \end{array}$$

For simplicity of the reduction rules we let $Q, R ::= (f_{ij} \mapsto s_{ij})^{j\in 1..n_p}$ range over request queues and identify mapping modulo reordering; $\alpha[f_i \mapsto s_i :: Q, b] :: C$ is a configuration containing the activity α which contains a request $f_i \mapsto a_i$, and $\alpha[Q, a] \in C$ means α is an activity of C with request queue Q and active object a ($\alpha[Q, a] \in C \Leftrightarrow \exists\, C'.\, C = \alpha[Q, a] :: C'$). Moreover, $\varnothing$ is the empty mapping, and $\mathrm{dom}(C)$ is the domain of the mapping C (i.e. the set of activities it defines). $noFV(s)$ is true if s has no free variables (the only binder being ς this definition is classical and straightforward). $\rightarrow_{\|}$is the parallel reduction on configuration defined in Table 1. We explain below each of the reduction rules:

- LOCAL performs a local reduction inside an activity: one step of ς-calculus reduction is performed on one request.
- ACTIVE creates an activity, the term passed as argument is the active object, it must not have free variables, that would escape the scope of their binder. The newly created activity must correspond to a fresh activity identifier, i.e. $\alpha \neq \gamma$ and γ is not defined in C. The newly created activity has an empty request queue, and the new activity identifier γ replaces the invocation to the *Active* primitive.
- REQUEST sends a request from an activity α to another activity β (by construction $\alpha \neq \beta$)). A new request is created at the destination, invoking the method l on the active object (t'); fresh future f_k is associated to this request, and replaces the invocation on the sender side.
- SELF-REQUEST is the particular case of the rule REQUEST where the destination is the sender ($\alpha = \beta$).
- REPLY updates a future: it picks the request calculating a value for the future f_k, and sends the current request value s back to the sender (or any other activity that refers to the future), this value may be any partially evaluated term. Note that necessarily $noFV(s)$ holds. The premise of the inference rule avoids having to deal separately with the case where $\alpha = \beta$.
- UPDATE-AO updates a field of an active object t'. It creates a new activity whom active object performs a (local) update on t' ($t'.l := \varsigma(x)s$).

$$\text{LOCAL}\quad \frac{s \rightarrow_\varsigma s'}{\alpha[f_i \mapsto s :: Q, t] :: C \rightarrow_\parallel \alpha[f_i \mapsto s' :: Q, t] :: C}$$

$$\text{ACTIVE}\quad \frac{\gamma \notin \mathrm{dom}(C) \cup \{\alpha\} \qquad noFV(s)}{\alpha[f_i \mapsto E[Active(s)] :: Q, t] :: C \rightarrow_\parallel \alpha[f_i \mapsto E[\gamma] :: Q, t] :: \gamma[\varnothing, s] :: C}$$

$$\text{REQUEST}\quad \frac{f_k \text{ fresh}}{\alpha[f_i \mapsto E[\beta.l] :: Q, t] :: \beta[R, t'] :: C \rightarrow_\parallel \alpha[f_i \mapsto E[f_k] :: Q, t] :: \beta[f_k \mapsto t'.l :: R, t'] :: C}$$

$$\text{SELF-REQUEST}\quad \frac{f_k \text{ fresh}}{\alpha[f_i \mapsto E[\alpha.l] :: Q, t] :: C \rightarrow_\parallel \alpha[f_k \mapsto t.l :: f_i \mapsto E[f_k] :: Q, t] :: C}$$

$$\text{REPLY}\quad \frac{\beta[f_k \mapsto s :: R, t'] \in \alpha[f_i \mapsto E[f_k] :: Q, t] :: C}{\alpha[f_i \mapsto E[f_k] :: Q, t] :: C \rightarrow_\parallel \alpha[f_i \mapsto E[s] :: Q, t] :: C}$$

$$\text{UPDATE-AO}\quad \frac{\gamma \text{ fresh} \qquad noFV(\varsigma(x)s) \qquad \beta[Q, t'] \in (\alpha[f_i \mapsto E[\beta.l := \varsigma(x).s] :: Q, t] :: C)}{\alpha[f_i \mapsto E[\beta.l := \varsigma(x).s] :: Q, t] :: C \rightarrow_\parallel \alpha[f_i \mapsto E[\gamma] :: Q, t] :: \gamma[\varnothing, t'.l := \varsigma(x).s] :: C}$$

Table 1: $\mathrm{ASP}_{\mathrm{fun}}$ semantics

3.5 Well-formed Configuration

To prove basic correctness of the semantics, we define a well-formed configuration as referencing only existing activities and futures.

Definition 1 (Well-formed configuration) *A configuration C is well-formed, denoted $wf(C)$ if and only if for all α, f_i, s, Q, and t:*

$$\alpha[f_i \mapsto s :: Q, t] \in C \Rightarrow \begin{cases} (\exists E.(s = E(\beta) \vee t = E(\beta))) \Rightarrow \beta \in \mathrm{dom}(C) \\ (\exists E.(s = E(f_k) \vee t = E(f_k))) \Rightarrow (\exists\, \gamma, R, t'.\, \gamma[R, t'] \in C \wedge f_k \in \mathrm{dom}(R)) \end{cases}$$

And we prove that any reduction preserves well-formedness of configurations.

Property 1 (Reduction preserves well-formedness)

$$(s \rightarrow_\parallel t \wedge wf(s)) \;\Rightarrow\; wf(t)$$

3.6 Initial Configuration

In most distributed languages, programmers do not write configurations, but usual programs with additional primitives, this is reflected by the $\mathrm{ASP}_{\mathrm{fun}}$ syntax given in Section 3.2. A "program" is a term a given by this static syntax. In

order to be evaluated, this program must be placed in an initial configuration of the form: $\alpha[f_0 \mapsto a, []]$. Note that this configuration is necessarily well-formed, and the activity α will never be accessible.

4 Typing Active Objects

This section provides a type-system for ASP_{fun}. First, our objective is to design a type-system for ASP_{fun}, which involves typing the Active primitive, but also type-checking an ASP_{fun} configuration. Then, two questions arise: first, is the type system correct, i.e. does it ensure subject reduction, second, is the type system decidable. This second question is trivially answered in the ς-calculus by typing all objects. Moreover the fact that the type system is formalised in Isabelle/HOL (see Section 5) using only inductive definitions, implicitly proves that there is an algorithm for deciding the typability relation: from any inductive definition in Isabelle/HOL we can automatically generate executable ML-code. But a type system is mainly interesting for the properties ensured. This one guarantees *type uniqueness*, and *well-formedness* of configurations, and more importantly *progress* (absence of dead-locks).

4.1 Type System for the ς-calculus

In this section we very briefly recall (a slightly adapted version of) the definition of the simple type system that Abadi and Cardelli devised as $\mathbf{Ob_1}$ in [2].

$$\textsc{Val}\ x \quad \frac{x \in \text{dom}(T)}{T \vdash x : T(x)} \qquad \textsc{Object Formation} \quad \frac{\vdash B_i \qquad \forall i \in 1..n \qquad (l_i \text{ distinct})}{T \vdash [l_i : {B_i}^{i \in 1..n}]}$$

$$\textsc{Type Object} \quad \frac{x_i : A :: T \vdash b_i : B_i \qquad \forall i \in 1..n \qquad \text{where } A = [l_i : {B_i}^{i \in 1..n}]}{T \vdash [l_i = \varsigma(x_i : A) {b_i}^{i \in 1..n}] : A}$$

$$\textsc{Type Call} \quad \frac{T \vdash a : [l_i : {B_i}^{i \in 1..n}] \qquad \forall j \in 1..n}{T \vdash a.l_j : B_j}$$

$$\textsc{Type Update} \quad \frac{T \vdash a : A \qquad x : A :: T \vdash b : B_j \qquad \forall j \in 1..n \qquad \text{where } A = [l_i : {B_i}^{i \in 1..n}]}{T \vdash a.l_j := \varsigma(x : A) b : A}$$

A and B range over types. The variable T represents a *type environment* containing type assumptions for variables. A type environment is a mapping from variables to types, its extension by an new assumption of x has type T is denoted by $T :: x : A$ (which supposes $x \notin T$); and rule $\textsc{Val}\ x$ accesses to the environment. The rule $\textsc{Object Formation}$ ensures the correct formation of object types. As an object type is a finite mapping from labels to types, the rule only ensure the finiteness of each type. $\textsc{Type Object}$ describes how an object's type is checked from its constituents: an object of type $[l_i : {B_i}^{i \in i..n}]$ is formed from bodies b_i of types B_i that may use the self parameter x_i. When a

method l_j is invoked on an object a of type $[l_i : B_i{}^{i\in i..n}]$ the result $a.l_j$ has type B_j (Type Call). Similarly an update of a method may take place in a position l_j of an object that has the right body type under the assumption of the self parameter (Type Update). In [2], additional rules ensure that the environment is well-formed, we simplified it here by defining environment as a mapping, and checking that added variables are fresh ones, possibly using equivalence modulo renaming of variables.

4.2 A Type System for $\mathbf{ASP_{fun}}$

The type system for ASP_{fun} is based on an inductive typing relation on ASP_{fun} terms. The typing rules for the ς-calculus can be adopted almost one to one for ASP_{fun}. The only difference is that In addition to typing of variables, we need to add types for futures and activities. Thus, we integrate a pair of parameters $\langle \Gamma_{act}, \Gamma_{fut} \rangle$ in the assumptions of a typing statement, i.e. we write $\langle \Gamma_{act}, \Gamma_{fut} \rangle, T \vdash x : A$ instead of just $T \vdash x : A$. These parameters consist of a mapping Γ_{act} from activities to the type of their active object, and another one Γ_{fut} from future identifiers to the type of the corresponding request value.

In a first step, we add to the four rules of the previous section (adorned with the additional parameters) the following three rules for local typing of ASP_{fun}.

Type Active
$$\frac{\langle \Gamma_{act}, \Gamma_{fut} \rangle, T \vdash a : A}{\langle \Gamma_{act}, \Gamma_{fut} \rangle, T \vdash \mathit{Active}(a) : A}$$

Type Activity Reference
$$\frac{\beta \in \text{dom}(\Gamma_{act})}{\langle \Gamma_{act}, \Gamma_{fut} \rangle, T \vdash \beta : \Gamma_{act}(\beta)}$$

Type Future Reference
$$\frac{f_k \in \text{dom}(\Gamma_{fut})}{\langle \Gamma_{act}, \Gamma_{fut} \rangle, T \vdash f_k : \Gamma_{fut}(f_k)}$$

These first three rules ensure the use of only defined references with respect to an environment, and define typing of the *Active* primitive. Next rule incorporates into a configuration the local typing assertions.

Type Configuration
$$\frac{\text{dom}(\Gamma_{act}) = \text{dom}(C) \qquad \text{dom}(\Gamma_{fut}) = \bigcup\{\text{dom}(Q) \mid \exists\ \alpha, a.\ \alpha[Q,a] \in C\} \qquad \forall \alpha, Q, a, C'.\ C = \alpha[Q,a] :: C' \Rightarrow \begin{cases} \langle \Gamma_{act}, \Gamma_{fut} \rangle, \varnothing \vdash a : \Gamma_{act}(\alpha) \ \wedge \\ \forall\ f_i \in \text{dom}(Q).\ \langle \Gamma_{act}, \Gamma_{fut} \rangle, \varnothing \vdash Q(f_i) : \Gamma_{fut}(f_i) \end{cases}}{\vdash C : \langle \Gamma_{act}, \Gamma_{fut} \rangle}$$

This rule states that a configuration C has the configuration type $\langle \Gamma_{act}, \Gamma_{fut} \rangle$ if the following conditions hold. The activity names defined in the configuration C and its activity type Γ_{act} are the same; the set of all future references defined in an activity of C and in Γ_{fut} are the same. And, for any activity of C, its active object a is well-typed with the typed defined in Γ_{act} for this activity, and each request is well-typed with the typed defined in Γ_{fut} for the future corresponding to this request. One could relate activity or future references to reference types [17], but also to typing rules for futures [15].

As a first interesting property, each expression in ASP_{fun} has a unique type.

Theorem 1 (Unique Type)

$$\langle \Gamma_{act}, \Gamma_{fut} \rangle, T \vdash a : A \wedge \langle \Gamma_{act}, \Gamma_{fut} \rangle, T \vdash a : A' \Longrightarrow A = A'$$

4.3 Subject Reduction

Subject reduction ensures that evaluation given by the reduction relation preserves the typing relation. Therefore it is often also called *preservation*. We prove subject reduction of ASP_{fun} with respect to the type system given in the previous section. In addition we prove that a well-typed term is well-formed.

We prove first the subject reduction property for the local reduction, i.e. for the ς-calculus.

Proposition 1 (Local Subject Reduction)

$$\langle \Gamma_{act}, \Gamma_{fut} \rangle, T \vdash t : A \ \wedge \ t \rightarrow_{\varsigma} t' \Rightarrow \langle \Gamma_{act}, \Gamma_{fut} \rangle, T \vdash t' : A$$

Then, we prove subject reduction for the full typing relation of configurations.

Theorem 2 (Subject Reduction)

$$\vdash C : \langle \Gamma_{act}, \Gamma_{fut} \rangle \ \wedge \ C \rightarrow_{\|} C' \Rightarrow \exists \ \Gamma'_{act}, \Gamma'_{fut} \ . \ \vdash C' : \langle \Gamma'_{act}, \Gamma'_{fut} \rangle$$

Interestingly, we also prove that typing ensures well-formedness:

Proposition 2

$$\vdash C : A \Rightarrow wf(C)$$

4.4 Progress and Absence of Deadlocks

Finally we can prove progress for the type system. Progress states that any expression of the language is either a value, i.e. cannot be evaluated any further, or can be reduced. For ASP_{fun}, we prove that either every request is a value

Theorem 3 (Progress)

$$\vdash C : \langle \Gamma_{act}, \Gamma_{fut} \rangle \ \wedge \ \alpha[f_i \mapsto a :: Q, t] \in C \Rightarrow isvalue(a) \vee \exists \ C' \ . \ C \rightarrow_{\|} C'$$

where $isvalue(a)$ is true if a is either an object $[l_i = \varsigma(x_i : A) b_i^{i \in 1..n}]$ or a reference to an activity. Indeed, both objects and activity references can only be considered as totally evaluated terms, and evaluated results.

By proving progress we also show that ASP_{fun} is deadlock free: as any term that is not already a value must progress, this ensures the absence of deadlock.

5 A Mechanised Proof in Isabelle

We formalised the entire theory presented in this paper in the interactive theorem prover Isabelle/HOL [16]. All presented results have been mechanically verified. This is a considerable effort but, as we believe, a necessary prerequisite for the development of complex languages. In this section we will give an outline of this mechanisation. The relevant Isabelle/HOL sources are available

from the authors web-page. Where appropriate we depict examples from the formalisation explaining necessary Isabelle/HOL syntax on the fly.

Isabelle/HOL is the most often used instantiation of the generic theorem prover Isabelle. Isabelle/HOL offers a classical higher order logic as a basis for the modelling of application logics. It has been successfully used in the analysis of type safety of Java and its Virtual Machine [18, 14]. Besides a sophisticated proof automation to help users develop interactive proofs it offers strong support for user defined concrete syntax enabling very natural notations. Inductive definitions and datatype definitions close to programming language syntax are sophisticated support methods for reasoning about programming language semantics. In particular datatype definitions are natural and also efficient. Whenever possible it is sensible to employ this feature for modelling as a datatype comes equipped with induction schemes and various properties like injectivity theorems of the constructors. All these properties are provided automatically by Isabelle/HOL. In addition semantic properties over datatypes are most often expressed in a clear manner using primitive recursion which is supported with powerful proof automation using rewriting techniques.

The semantics for the ς-calculus in Isabelle/HOL has been defined in [12], we proved its equivalence with the shorter version presented in Section 3.1. The formalisation of the ς-calculus in Isabelle/HOL presents a formal model of objects and its operational semantics based on DeBruijn indices, a parallel reduction relation for objects, the proof of confluence for the theory of objects reusing as much as possible Nipkow's HOL-framework for the lambda calculus. As a further evolution of [12] we have meanwhile extended this basic formalisation with a simple type system and proved type safety for the ς-calculus. This basic theory is now extended to support ASP_{fun} as we are going to outline in the current section.

5.1 Defining ASP_{fun}

The formalisation of functional ASP is constructed as an extension of the base Isabelle/HOL theory for the ς-calculus. The term type of the ς-calculus is represented by an Isabelle/HOL datatype definition called `dB`. According to the minimal extension of ASP_{fun} we extend this basic datatype by three additional fields for activities, references to activities, and references to futures. The resulting datatype for basic terms of ASP_{fun} is then as follows.

```
datatype dB =
    Var nat
  | Obj Label ⇒f dB
  | Call dB Label
  | Upd dB Label dB
  | Active dB
  | ActRef ActivityRef
  | FutRef FutureRef
```

In this datatype definition Objects are represented as finite maps marked by the type constructor $\Rightarrow_f$ we developed especially for this purpose. [1] It is crucial to have this basic type to be able to employ the Isabelle/HOL datatype construction here because the Object constructor of the type `dB` is recursive. Although

[1] Surprisingly the standard Isabelle/HOL library does not offer finite maps.

the semantics of activities is based on the local evaluation of basic terms, it relies on parallel configurations. Therefore at the level of local term reduction, there are no substantial rules added for the three new constructors apart from standard context reduction rules. Consequently, the proof of confluence already performed for the ς-calculus could be upgraded with only minor changes to the local reduction of terms.

In the next step we modelled configuration of activities. To this end, we use simply partial functions (expressed by the type constructor $\not\Rightarrow$) from activity names to activities at the surface and partial functions from future names to futures inside the local structures. As configurations are functions the order does not matter, as they are partial they may be arbitrarily extended. Hence the following partial function types in Isabelle/HOL are a natural basis for the representation of configurations.

```
futmap = FutureRef ⇀ dB
configuration = ActivityRef ⇀ (futmap × dB)
```

To give some more flavour of the complexity of the Isabelle/HOL model we depict the following central rule LOCAL for local reduction and explain it in detail below. This rule is part of an inductive definition for the reduction relation $\rightarrow_{\|}$ on configurations. An inductive definition in Isabelle/HOL defines a set, here the relation $\rightarrow_{\|}$, by a set of simple rules. The set defined by an inductive definition is the least set that is closed under those rules. The rules of the inductive definition for $\rightarrow_{\|}$ are exactly the corresponding rules ACTIVE, REQUEST (see also below), SELF-REQUEST, REPLY, and update of the active object (UPDATEAO).

```
local: ⟦ a →β a'; C A = Some (m,a''); m(fk) = Some a ⟧
       ⟹ C →‖ C (A ↦ (m (fk ↦ a'), a''))
```

The assumption of the rules – enclosed in Isabelle/HOL's meta-logic brackets ⟦⟧ – presuppose that a term `a` locally reduces to term `a'` and that a given configuration `C` contains for name `A` the future map/activity pair `(m,a'')` — in Isabelle/HOL a defined point in a partial function is described by the option constructor `Some`. Furthermore, the rule for local reduction of configurations assumes that the previous local term `a` is contained in the future map at position `fk`, i.e., `a` is a future. Given the setup described by these assumptions, the next configuration is defined from the current configuration `C` by replacing the activity `A` with an updated value in the future map `m` at position `fk`. Update of partial maps is quite concisely annotated using ↦ as an infix operator.

The other rules are expressed in a similar fashion. However, for the natural expression of these semantic rules we need a particular feature commonly used in semantic description, but not easily expressed in Isabelle/HOL: the "hole"-notation as introduced in Section 3.1. The context expresses context information as a prerequisite for a rule application. An example where this feature is used it the rule for request (see Section 3.4).

In our Isabelle/HOL model we developed an elegant mechanisation of a "context" using again the datatype feature that we have already seen above as a means to express `dB` terms. Isabelle/HOL internally generates rules for a datatype specification, most notably induction rules for recursive types and injectivity rules for the constructors. Pattern matching facilitates case analysis proofs crucial for reasoning with complex languages.

```
datatype general_context =
    cHole
  | cObj FmapLabel general_context
  | cCall general_context Label
  | cUpdL general_context Label dB
  | cUpdR dB Label general_context
  | cActive general_context;
```

In this original representation of contexts by a specific datatype constructor we exploit the power of the efficient datatype feature of Isabelle while at the same time finding a first class representation of the syntactical concept of "context". For the use of contexts we define an operator to "fill" the "hole" enabling a fairly natural notation of `E↑t` for $E[t]$.

```
consts Fill :: [general_context, dB]  ⇒  dB ("↑")
```

Functions over datatypes may be defined in a particularly efficient way in Isabelle/HOL using primitive recursion. Efficient means in this context that proofs involving these operators may be mostly solved automatically using automatic rewriting techniques provided in Isabelle. The semantics of the `Fill` operator is described by the following set of equations.

```
primrec
  Fill cHole x = x
  Fill  (cObj f E ) x =  Obj ((FLmap f)((FLlabel f)↦(Fill E x)))
  Fill  (cCall E l) x =  Call (Fill E x) l
  Fill (cUpdL E l (y::dB)) x = Upd (Fill E x) l y
  Fill (cUpdR (y::dB) l E) x = Upd y l (Fill E x)
  Fill (cActive E) x = Active (Fill E x)
```

We can illustrate the benefits of our context concept most directly by the Isabelle/HOL representation of the configuration rule for requests.

```
request: ⟦ ∀ A∈ dom c. fn ∉ dom(fst(the(C A)));
          C A' = Some(m',a'); m'(fk) = Some(E↑(Call(ActRef B) li);
          C B = Some(mb, a);A'≠ B ⟧
          ⟹ C →‖ C (A'↦ (m' (fk ↦ (E↑(FutRef(fn)))), a'))
                 (B ↦ (mb (fn ↦ (Call a li)), a))
```

Here we can very naturally express that if in the current configuration `c` a future `fk` of activity `A'` holds a term that contains syntactically a call to some other activity `B`, then this request can be dealt with by replacing the request subterm `Call(ActRef B) li` in the caller context by "fresh" future reference `fn` and queueing the request into activity B's future list. Similarly we can encode the other rules to provide a small step operational semantics given by this inductively defined $\rightarrow_{\|}$ relation.

5.2 Proving Correctness of the Reduction

Based on the formalisation of a configuration and the extended small-step operational semantics we first prove that the well-formedness of a configuration is preserved by the reduction relation.

When comparing to the definition of well-formedness as given in Section 3.5 the Isabelle version of these properties is straightforward, based on the

definition of configuration as seen in the previous section. For example the first well-formedness property `wf_1a` reads in Isabelle/HOL as follows.

```
∀ A ∈ dom C. ∀ d m E B. C A = Some(m,d) ∧ d = E↑(ActRef B) ⟶ B ∈ dom C
```

The other three properties are translated into Isabelle/HOL in the same straightforward way. We then simply define well-formedness as the conjunction of the four properties.

```
wf_conf :: configuration ⇒ bool
wf_conf C == wf_1a C ∧ wf_1b C ∧ wf_2a C ∧ wf_2b C
```

We prove subject reduction formally in Isabelle/HOL. More precisely we prove the following properties based on the above definition.

```
C →‖ C' ⟶ (wf_1a C ∧ wf_1b C  ⟶ wf_1a C' ∧ wf_1b C')
C →‖ C' ⟶ (wf_2a C ∧ wf_2b C  ⟶ wf_2a C'∧ wf_2b C')
```

The proofs of the wellformedness preservation properties are about 1100 lines of Isabelle/HOL proof script code. However, one has to count here the additional 1000 lines of proofs we performed on the theory infrastructure (lemmata) for the general contexts. They form the basis for the definition of the configuration's small step semantics. Corresponding lemmata are prerequisite for the proof of the wellformedness properties. The proofs of the basic properties of context are often tricky as they involve simultaneous induction over the terms caused by the partiality of the map structure of objects that enforce the consideration of definedness. The wellformedness properties are difficult in theory already and technically the complexity of the involved functional structure adds to the challenge.

Once we have defined typing – which we will do shortly in the next section – we will be able to use well-formedness to extend correctness to the type system. We formally proved in Isabelle the following theorem.

```
theorem Typing_implies_WF: ⊢ C : B ⟹ wf_conf C
```

5.3 Typing and Progress

We define a typing relation that inductively builds up term-to-type pairs with respect to two parameters: one is a configuration and one is a type environment. This corresponds to the two-layered structure of base terms and configurations of ASP_{fun}.

```
typing :: (Ctype × (type list) × dB  × type) set
```

where the two parameter types are the following datatypes.

```
datatype type =  Object (Label ⇒f type)
datatype Ctype = TConfig (ActivityRef ⇻ type)(FutureRef ⇻ type)
                                        ("⟨ _,_ ⟩")
```

We define the standard infix syntax for typing statements.

```
C, E ⊢  a : A
```

which is just syntactic sugar for the inhabitance of the previously defined relation.

```
(C, E, a, A) ∈ typing
```

Then the relation `typing` is defined using an Isabelle/HOL *inductive definition*. The rules of the inductive definition are exactly the typing rules for ASP_{fun} introduced in Section 4. For comparison we show just the rule Type Object

```
⟦ dom b = do B; ∀ l ∈ do B. C, E<0:B> ⊢ the(b l): B!l ⟧
   ⟹ C, E ⊢ Obj b B: B
```

The second layer of typing characterises the relationship between the first configuration parameter and the actual configuration by enforcing that the locally well-typed terms adhere to the definedness conditions of the first configuration parameter `C`. To encode this in Isabelle/HOL we first define a second inductive set that expresses typing without assumptions, i.e. no parameters.

```
Ctyping :: (configuration × Ctype) set
```

We overload the same operator `⊢ C: T` to abbreviate `C, T ∈ Ctyping`. The complex rule Type Configuration encoding the semantic well-formedness on configurations in Isabelle/HOL is as follows.

```
⟦ dom fa  = dom C; dom ff = Union{ dom m | m. ∃ A a. C A = Some (m,a)};
 ∀ A m a . C A = Some (m,a) ⟶ ⟨ fa, ff⟩,[]⊢ a:the(fa A) ∧
 ∀ fi ∈ dom m. ⟨ fa, ff⟩,[] ⊢ the(m fi):the(ff fi) ⟧
  ⟹ ⊢ C: ⟨ fa, ff ⟩
```

We completely proved in Isabelle/HOL Theorems 2 and 3.

```
theorem Csubject_reduction:
   ⊢ C : B ⟹ (∀ C' . C →∥ C' ⟶ ∃ B'. ⊢ C' : B')
theorem progress_ASP :
   ⟦ ⊢ C: B; C A = Some (m,a); m fi = Some t; ¬ (isvalue t) ⟧
    ⟹ ∃ C'. C →∥ C'
```

6 Conclusion

The formalisation in Isabelle/HOL adds the necessary quality assurance to a language development where rules and properties are intricate while the need for verification is as worthwhile as imperative.

Altogether the entire development takes just under 10000 lines of Isabelle/HOL code. The line numbers of proofs do not necessarily reflect the difficulty and the amount of time that went into a definition or a proof. Once a good model is found proofs become simple. The most difficult steps were the first model for object terms, the translation of common semantic notions and rules into Isabelle/HOL. The progress proof was harder than subject reduction although the former is much longer.

We have presented here a functional calculus for distributed activities. This calculus has several nice properties. The design of the distributed calculus allows to return future evaluations before the end of their computation. The absence of dead locks comes from this functional nature. It avoids blocking.

The absence of message not understood is due to typing. Additionally distributed reduction behaves well with respect to typing. We have additionally shown that well formedness is ensured by typing.

The proof of confluence is what we envisage as our ultimate goal. Therefore, we are currently working on the definition of a parallel evaluation relation that will enable us to reduce the confluence property of the original reduction of configurations to the diamond property of this parallel reduction. The current difficulty is to generalize the notion of reduction contexts such that parallel evaluation can be expressed.

References

[1] Martín Abadi and Luca Cardelli. An imperative object calculus. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT*, volume 915 of *Lecture Notes in Computer Science*, pages 471–485. Springer, 1995.

[2] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[3] Gul Agha. An overview of actor languages. *ACM SIGPLAN Notices*, 21(10):58–67, 1986.

[4] D. Caromel, L. Henrio, and B. Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–134. ACM Press, 2004.

[5] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Object*. Springer-Verlag, 2005.

[6] A. Ciaffaglione, L. Liquori, and M. Miculan. Reasoning about object-based calculi in (co)inductive type theory and the theory of contexts. *JAR, Journal of Automated Reasoning*, 39:1–47, 2007.

[7] Tom Van Cutsem, Jessie Dedecker, and Wolfgang De Meuter. Object-oriented coordination in mobile ad hoc networks. In Amy L. Murphy and Jan Vitek, editors, *COORDINATION*, volume 4467 of *LNCS*, pages 231–248. Springer, 2007.

[8] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *ESOP*, pages 316–330, 2007.

[9] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-oriented programming in ambienttalk. In Dave Thomas, editor, *ECOOP*, volume 4067 of *LNCS*, pages 230–254. Springer, 2006.

[10] Cormac Flanagan and Matthias Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9(1):1–31, 1999.

[11] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[12] Ludovic Henrio and Florian Kammüller. A mechanized model of the theory of objects. In *9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, LNCS. Springer, June 2007.

[13] Alan Jeffrey. A distributed object calculus. In *ACM SIGPLAN Workshop Foundations of Object Oriented Languages*, 2000.

[14] G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, April 2002.

[15] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, November 2006.

[16] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[17] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, March 2002.

[18] D. v. Oheimb and T. Nipkow. Machine-checking the java language specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 119–156. Springer-Verlag, 1999.

[19] Akinori Yonezawa, Etsuya Shibayama, Toshihiro Takada, and Yasuaki Honda. Modelling and programming in an object-oriented concurrent language ABCL/1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 55–89. MIT Press, Cambridge, Massachusets, 1987.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399